UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

19|20

Trabajo fin de grado

Herramientas modernas en Redes Neuronales: la librería PyTorch

Santiago Cerezo Sánchez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

excelencia Campus Internacional UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Herramientas modernas en Redes Neuronales: la librería PyTorch

Autor: Santiago Cerezo Sánchez
Tutor: José Ramón Dorronsoro

junio 2020

Santiago Cerezo Sánchez
Herramientas modernas en Redes Neuronales: la librería PyTorch

Santiago Cerezo Sánchez
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia y amigos

El sabio no dice nunca todo lo que piensa, pero siempre piensa todo lo que dice.

Aristóteles

Agradecimientos

En primer lugar quiero agradecerle a mi madre y a mi padre por haberme dado la oportunidad de cursar la carrera que quería y por apoyarme a lo largo de estos años para que pudiera sacar adelante mis estudios. También a mi hermana, por haberme sabido animar cuando lo he necesitado y a mis amigos, a los de toda la vida y a los que he hecho durante mis años en la carrera. Por último pero no menos importante también me gustaría agradecerle a mi tutor José Ramón Dorronsoro por la oportunidad de hacer este TFG.

Resumen

Este TFG trata sobre uno de los temas que actualmente está en constante desarrollo, la inteligencia artificial, más concretamente las redes neuronales.

Las redes neuronales van desde las más sencillas, como pueden ser el perceptrón de Rosenblatt , hasta otras más complejas como las redes convolucionales, que se usan normalmente para la clasificación de imágenes.

Para la definición y entrenamiento de estas redes neuronales existen diferentes librerías en Python que facilitan la tarea. Una de estas es la librería PyTorch, de Facebook, que será la que se estudie en este TFG. Haciendo uso de esta librería se verá como definir, entrenar y evaluar distintos tipos de redes neuronales. Al igual que antes se empezará por las más sencillas para finalmente llegar a las redes convolucionales.

Finalmente se aplicará estos conocimientos a la resolución de problemas típicos, y se comparará la librería PyTorch con Keras a la hora de entrenar la red neuronal.

Palabras clave

PyTorch, red neuronal, Keras

Abstract

This TFG is about a field that is constantly in development, the artificial inteligence, especifically about neural networks.

The neural networks go from the most simple ones, like the Rosenblatt perceptron to others that are more complex, like the convolutional networks, that are normally used for solving problems that involve images.

For the definition and training of neural networks there are some libraries for Python, that make this task easier. One of those is the PyTorch library, of Facebook, witch is the one that is going to be used in this TFG. Making use of this library we will see how to create, train and evaluate diferent neural networks. This process goes from the most simple neural network to the convolutional network.

Finally there will be some tipical problems solved using neural networks and there will be a comparision about the time that takes to train a neural network in PyTorch and in Keras.

Keywords

PyTorch, neural network, Keras

Índice

Listas

Lista de figuras

Lista de tablas

1

Introducción

1.1. Motivación

El tema principal sobre el que trata este trabajo de fin de grado es el aprendizaje automático, concretamente las redes neuronales. Esta rama de la informática se basa en cómo funcionan las neuronas naturales de nuestro cerebro para aprender. Las redes neuronales intentan modelar este proceso para conseguir resolver distintos problemas. Este proceso de intentar simular el comportamiento de una neurona en una máquina ha dado lugar a distintas aproximaciones a lo largo de los años. Las primeras de estas aproximaciones se dieron durante los años 50 y se han ido desarrollando a lo largo de los años haciendo que hoy en día sean una de las herramientas más útiles dentro de la Inteligencia Artificial.

La principal ventaja es que permiten resolver problemas muy diversos de manera relativamente sencilla. Por ejemplo: problemas de predicción de valores, de clasificación de datos o de imágenes. Estos últimos combinan las redes neuronales con los filtros, lo cual permite destacar ciertas características de una imagen.

Todas estas aplicaciones resultan muy útiles en la sociedad actual, lo cual hace que sea un campo en constante evolución y que ha sido fundamental en los últimos años, y que seguirá siéndolo en el futuro ya que muchos de los planes futuros como la llegada del ser humano a Marte o los coches autónomos se basan en la Inteligencia Artificial.

El principal inconveniente de las redes neuronales es que para problemas reales se tarda mucho tiempo en entrenar la red, pero este problema se podría solucionar en el futuro si se consigue aumentar la capacidad de cómputo, por ejemplo, avanzando en la computación cuántica.

En los últimos años se han desarrollado diversas herramientas que permiten diseñar y entrenar los distintos tipos de redes neuronales, facilitando la tarea de los programadores a la hora de valerse de estos sistemas para resolver problemas. Ejemplos de estas librerías son: TensorFlow, una librería de Google, Keras, que es una librería de más alto nivel que se construye sobre TensorFlow y finalmente la que ha sido objeto de estudio en este TFG, PyTorch, que es de Facebook. Todas estas librerías están

escritas en Python, que es un lenguaje de alto nivel orientado a objetos.

Tanto TensorFlow como PyTorch son librerías que permiten trabajar con las redes neuronales a muy bajo nivel, es decir, dan mucha libertad a la hora de cómo definir la redes neuronales, lo cual hace que sea más costosa la codificación del código de la red neuronal, pero por otra parte, permite definir dicha red de manera más precisa.

Keras tiene una aproximación de más alto nivel, ocultando al programador la parte de más bajo nivel, la cual se apoya en TensorFlow. Esto hace que sea más fácil programar en ella pero limita la libertad del programador a la hora de diseñar la red neuronal.

1.2. Objetivos

Los objetivos a cumplir en este TFG son:

- Introducir la teoría matemática detrás de las redes neuronales.
- Ver la estructura y las funcionalidades que tiene la librería PyTorch y cómo aplicarlas a las redes neuronales
- Resolver algunos ejemplos típicos mediante la librería PyTorch.

1.3. Estructura

El TFG se organizará de la siguiente manera:

- **Redes neuronales**: En este capítulo se introduce la teoría detrás de las redes neuronales. Se sigue un proceso incremental empezando por redes con una sola neurona, siguiendo con sistemas con varias neuronas cuya salida es la entrada de otra capa de neuronas y acabando con redes neuronales convolucionales. Se introduce la teoría detrás del algoritmo de descenso por gradiente, que es el algoritmo que se usa para entrenar las redes neuronales.
- **La librería PyTorch**: En este capítulo se describe cómo se manejan los datos en PyTorch, cómo calcula los gradientes PyTorch y cómo definir y entrenar los distintos tipos de redes neuronales usando la librería PyTorch. También se cuenta el proceso que sigue PyTorch a la hora de aplicar el algoritmo de descenso por gradiente, que se usa para entrenar las redes neuronales.
- **Problemas resueltos con PyTorch**: En este capítulo se muestra como se resolverían los problemas de Boston Housing, Pima y MNIST, que se resuelven con los tres tipos principales de redes neuronales que se mencionan en este TFG y se muestran los resultados obtenidos con redes neuronales creadas con PyTorch.
- **Conclusiones**: En este capítulo se recogen las observaciones más importantes que se han dado en los capítulos anteriores y los posibles trabajos futuros sobre la librería PyTorch.

2

Redes neuronales

En este capítulo se contará la teoría de redes neuronales necesaria para los ejemplos que se desarrollarán en los capítulos 3 y 4. Se irá desde el perceptrón de Rossenblatt hasta la redes convolucionales, pasando por la redes neuronales multicapa.

El capítulo se organiza de la siguiente manera, en la sección 2.1 se hablará de las redes neuronales feedforward, tanto para regresión lineal, y regresión logística multicapa y se describirá el concepto de regularización, muy importante a la hora de entrenar redes neuronales. En la sección 2.2 se hablará de la teoría detras de las redes convolucionales, que se usan generalmente en problemas relacionados con imágenes.

Para desarrollar los conceptos teóricos de redes neuronales se han consultado diferentes fuentes de información que son: los libros Redes neuronales: conceptos fundamentales y aplicaciones a control automático [1] y Fundamentals of neural networks: architectures, algorithms, and applications [2].

2.1. Redes neuronales feedforward

Se hablará en primer lugar de la primera aproximación a lo que posteriormente se convertirían en las redes neuronales feedforward que conocemos hoy en día. Esta aproximación es el perceptrón de Rosenblatt. Éste es un clasificador que permite resolver problemas de clasificación homogéneos, que son aquellos cuyas clases son separables por un hiperplano de ecuación $\mathbf{y} = W \cdot \mathbf{x}$, eso es, un hiperplano que pasa por el origen.

Sea $S = \{(y^p, x^p)\}$ el conjunto de los datos introducidos a la hora de entrenar este clasificador, donde $y^p = 1$ o $y^p = -1$. La función que se utiliza para determinar la clase es:

$$\begin{cases} W \cdot \mathbf{x^p} \leq 0 & \text{si } y^p = -1 \\ W \cdot \mathbf{x^p} > 0 & \text{si } y^p = 1 \end{cases}$$

esto implica que para todos los datos de entrenamiento se quiere: $y^p \cdot W \cdot \mathbf{x^p} > 0$. El objetivo de

entrenar este clasificador es encontrar el vector W con el que la condición anterior se cumple. Para encontrar dicho vector se utiliza el algoritmo Delta de Rosenblatt que es el siguiente:

```
inicializar W=0
mientras condicion de parada no se cumple
        tomar siguiente dato de entrenamiento (x,y)
         si yWx<= 0 entonces
                W = W + yx
```

Este clasificador usa una función de error que se calcula como:

$$Err = -\sum_{\forall p: y^p \cdot W \cdot \mathbf{x^p} \leq 0} y^p \cdot W \cdot \mathbf{x^p}$$

Esta función no es diferenciable. Al usar esta función de error el algoritmo anterior es muy similar al algoritmo de descenso por gradiente que se usa en las redes neuronales actuales. El gradiente aproximado sería $-y^p \cdot \mathbf{x^p}$ si $y^p \cdot W \cdot \mathbf{x^p} \leq 0$ y 0 en el caso contrario. El no ser diferenciable hace que se necesiten usar algoritmos más complejos.

2.1.1. Regresión lineal

La regresión lineal es una nueva aproximación al problema planteado por el perceptrón de Rosenblatt en la cual ya no se buscan solo soluciones homogéneas, sino aquellas que sigan la ecuación $\mathbf{y} = W_0 + W \cdot x$. También se cambia la función de error usada por el perceptrón de Roseblatt por la del error cuadrático medio, que sigue la siguiente fórmula:

$$\hat{e}(W, W_0) = \frac{1}{2N} \times \sum_p (\mathbf{y^p} - W \cdot \mathbf{x^p} - W_0)^2$$

El objetivo es minimizar $\hat{e}(W, W_0)$ para los datos de nuestro problema. Para ello lo primero se centran x e y para tomar $W_0 = 0$ por tanto se trata de minimizar:

$$\hat{e}(W) = \frac{1}{2N} \times \sum_p (\mathbf{y^p} - W \cdot \mathbf{x^p})^2$$

Suponiendo que $\mathbf{Y}$ es el vector $N \times 1$ de las soluciones y que X es una matriz $N \times d$ siendo N el número de datos del conjunto S y d la dimensión de los datos de entrada de la regresión lineal tenemos que:

$$\hat{e} = \frac{1}{2N} \times (W^t \cdot X^t \cdot X \cdot W - 2W^t \cdot X^t \cdot Y + Y^t \cdot Y),$$

De aquí se deduce que la solución a $\nabla \hat{e}(W) = 0$ es:

$$W = (X^t \cdot X)^{-1} \cdot X^t \cdot Y,$$

suponiendo que $X^t \cdot X$ sea una matriz invertible.

En ocasiones este cálculo no se puede llevar a cabo debido al gran tamaño de los datos procesados por estos problemas. Por tanto se usa la aproximación de descenso por gradiente para obtener el valor de W que minimiza el error. Este método consiste en ir actualizando el vector W de manera iterativa de la siguiente forma:

$$W^k = W^{k-1} - \rho \nabla \hat{e}(W^{k-1}),$$

donde ρ es la tasa de aprendizaje y W^{k-1} es el vector de pesos del paso anterior. Si este algoritmo converge a W^*, entonces $\nabla \hat{e}(W^*) = 0$.

2.1.2. Regresión logística

Otra alternativa en el ámbito de la clasificación al problema planteado por el perceptrón de Rosenblatt es el uso de la función sigmoide, que tiene una salida con valores entre 0 y 1. La función sigmoide tiene la siguiente fórmula:

$$\sigma(z) = \frac{1}{1 + \exp -z},$$

En este caso la salida del perceptrón puede ser vista como una probabilidad a posteriori. Es decir, la probabilidad de que dado un vector de entrada x, la salida sea de clase 1. En la regresión logística se aplica la función sigmoide a la salida de la neurona. Por tanto la probabilidad a posteriori se puede expresar como:

$$P(1|x) = \frac{1}{1 + \exp -(W_0 + W \cdot x)}$$

Al tratarse de problemas con dos clases se deduce que:

$$P(0|x) = 1 - P(1|x) = \frac{1}{1 + \exp\left(W_0 + W \cdot x\right)}$$

La frontera de decisión la constituye el hiperplano formado por los puntos $W_0 + W \cdot x = 0$ que se corresponde con el caso donde $P(1|x) = P(0|x) = 0{,}5$. Al igual que para la regresión lineal se necesita definir una función de error, que nos permita evaluar cómo de bien se ajustan unos pesos para separar un conjunto de datos concreto.

Como en el caso de la regresión lineal, sea $S = \{(y^p, x^p)\}$ nuestra muestra de datos. Supongamos que los $\mathbf{Y} = \{\mathbf{y^p}\}$ se obtienen de manera independiente usando el modelo de regresión logística con pesos W y W_0 a partir de la matriz $\mathbf{X} = \mathbf{x^p}$. Entonces:

$$
\begin{aligned}
P(Y|X;W,W_0) &= \prod_{1}^{N} P(\mathbf{y^p}|\mathbf{x^p};W,W_0) = \{\prod_{\mathbf{y^p}=1} P(1|\mathbf{x^p})\}\{\prod_{\mathbf{y^p}=0} P(0|\mathbf{x^p})\} \\
&= \prod_{p=1}^{N} P(1|\mathbf{x^p})^{\mathbf{y^p}} P(0|\mathbf{x^p})^{1-\mathbf{y^p}}
\end{aligned}
$$

El objetivo es maximizar esta probabilidad. Como el logaritmo neperiano es una función creciente, aplicamos el logaritmo a la anterior probabilidad, obteniendo:

$$
\begin{aligned}
l(W,W_0;S) &= \log P(Y|X;W,W_0) \\
&= \sum_{p} \mathbf{y^p} \log P(1|\mathbf{x^p}) - \mathbf{y^p} \log P(0|\mathbf{x^p}) + \log P(0|\mathbf{x^p}) \\
&= \sum_{p} \mathbf{y^p} \log \frac{P(1|\mathbf{x^p})}{P(0|\mathbf{x^p})} + \sum_{p} \log P(0|\mathbf{x^p}) \\
&= \sum_{p} \mathbf{y^p}(W_0 + W \cdot \mathbf{x^p}) - \sum_{p} \log\left(1 + \exp\left(W_0 + W \cdot \mathbf{x^p}\right)\right)
\end{aligned}
$$

Para hallar los valores que maximizan esta función calculamos los que minimizan $-l(W,W_0;S)$. Esto se debe a que al ser $l(W,W_0;S)$ una función creciente al cambiar de signo se convierte en decreciente y por tanto el valor mínimo de esta función se corresponde con el máximo que buscamos, la fórmula por tanto es:

$$\hat{W_0^*}, \hat{W^*} = argmin_{\hat{W_0^*},\hat{W^*}}(-l(W,W_0;S))$$

Además la función $-l$ es diferenciable; por tanto bastará con resolver $\nabla l(W,W_0;S) = 0$. Esto no se puede resolver directamente, pero al igual que en el caso de la regresión lineal, se puede aproximar con métodos numéricos como el de descenso por gradiente.

2.1.3. Redes neuronales multicapa

Las regresiones lineal y logística se usan para resolver problemas donde los datos son básicamente separables por un hiperplano. Otros problemas que no sean separables por un hiperplano requieren el uso de redes neuronales con más capas. Cada una de las capas estará compuesta de una o más neuronas, como la explicada en la regresión lineal, a cuya salida se le puede aplicar la función sigmoide o alguna otra función de activación como son la función ReLU o la función tangente hiperbólica.

El caso más sencillo es el de la regresión lineal con una sola capa oculta. Al igual que antes sea $S = \{(y^p, x^p)\}$ un conjunto de datos. A continuación definimos la notación que se usará para referirnos a los pesos y el bias (es una traslación del hiperplano para que no tenga por que pasar por el origen) de las distintas capas. Sea una red neuronal con una capa oculta de H neuronas, una entrada de tamaño D y una salida de tamaño 1. Si o^h son las salidas de las neuronas de la capa oculta entonces:

$$o^h = \sigma(W_{h0}^H + W_h^H \cdot \mathbf{x^p}),$$

donde σ denota una de las funciones de activación antes mencionada y W_{h0}^H es el bias de la neurona h de la capa oculta y W_h^H son los pesos de dicha neurona. Por tanto la salida de la red neuronal será:

$$\hat{y} = W_0^O + W^O \cdot \mathbf{o^H},$$

donde o^H es el vector formado por los distintos o^h calculados anteriormente y W_0^O es el bias de la capa de salida y W^O son los pesos de la capa de salida. A la hora de entrenar una red neuronal con una o varias capas ocultas se aplica la regla de la cadena para calcular el gradiente de los pesos y los bias de cada capa de la red para así poder aplicar el algoritmo de descenso por gradiente. Para el caso de la regresión lineal con una capa oculta usamos la función de pérdida de error cuadrático medio. Los gradientes de los pesos serían:

$$\begin{aligned} \frac{\nabla e(W)}{\partial W^O} &= (\hat{y} - y) \times \frac{\partial \hat{y}}{\partial W^O} = (\hat{y} - y) \times \mathbf{o^H} \\ \frac{\partial e(W)}{\partial W_0^O} &= (y - \hat{y}) \times \frac{\partial \hat{y}}{\partial W_0^O} = (y - \hat{y}), \end{aligned}$$

en la capa de la salida y en la capa oculta son:

$$\begin{aligned} \frac{\nabla e(W)}{\partial W_h^H} &= \frac{\partial e(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial o^h} \times \frac{\partial o^h}{\partial W_h^H} \\ &= (\hat{y} - y) \times W^O \times \sigma' \times \mathbf{x^p} \\ \frac{\partial e(W)}{\partial W_{h0}^H} &= \frac{\partial e(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial o^h} \times \frac{\partial o^h}{\partial W_{h0}^H} \\ &= (\hat{y} - y) \times W_0^O \times \sigma' \end{aligned}$$

Estos son los gradientes que posteriormente se usan para actualizar los pesos en cada iteración del algoritmo de descenso por gradiente.

Al ser posible que la función de pérdida tenga más de un mínimo, surge el problema de que para alcanzar una solución óptima que minimice el error, hay que inicializar de manera adecuada los pesos y los bias. Esto se debe a que el algoritmo de descenso por gradiente puede encontrar un mínimo local de la función de pérdida y no el mínimo global.

En el caso de la regresión logística, a la salida de la red neuronal se le aplica la función sigmoide; por tanto el entrenamiento es igual que en el caso de la regresión lineal pero añadiendo la derivada de la función sigmoide en función de la salida de la regresión lineal.

2.1.4. Regularización

El vector óptimo de la regresión $W = (X^t \cdot X)^{-1} \cdot X^t \cdot Y$ solo se puede calcular si $X^t \cdot X$ es una matriz invertible, lo cual no sucede si algunos de los datos de entrada están correlacionados. Este

problema se resuelve invirtiendo $X^t \cdot X + \alpha \cdot I$ para algún $\alpha > 0$. Se puede demostrar que la solución $W = (X^t \cdot X + \alpha \cdot I)^{-1} \cdot X^t \cdot Y$ minimiza:

$$\hat{e_R}(W) = \frac{1}{2N} \times \sum_p (\mathbf{y^p} - W \cdot \mathbf{x^p})^2 + \frac{\alpha}{2} \cdot ||W||^2$$

El α adecuado se encuentra mediante validación cruzada al entrenar la red neuronal. En el caso de las redes multicapa el proceso de regularización es obligatorio ya que permite controlar el sobre-ajuste de la red a la hora de entrenarla.

El nuevo gradiente será por tanto:

$$\nabla \hat{e_R} = \nabla \hat{e} + \alpha \cdot W$$

2.2. Redes Convolucionales

Las redes convolucionales son redes neuronales multicapa que se usan principalmente en problemas de clasificación de imágenes. Para ello cuentan con dos tipos de capas especiales, que son las capas convolucionales y las capas de pooling. Estas capas se aplican sobre las imágenes para obtener un vector de datos que sirva como entrada para una red de regresión logística como las vistas en la sección anterior.

Las capas de convolución le aplican una serie de filtros a los datos de entrada, que son generalmente imágenes, los cuáles resaltan ciertas características que permiten distinguirlas posteriormente a la hora de clasificar dichas imágenes. Estas capas aplican una operación matemática que es la convolución de matrices, ya que las imágenes pueden verse como matrices de datos cuando se representan en formato digital. Esta operación se puede ver como la magnitud en la que se superponen la matriz de píxeles y la matriz que constituye el filtro trasladada y rotada.

Para los píxeles que se encuentran en los bordes de la imagen se suele aplicar lo que se denomina padding, que consiste en rellenar con ceros los laterales de la imagen para así poder calcular el valor de la convolución en los pixeles de los bordes.

Una vez aplicada la capa de convolución, para reducir el tamaño de la salida, se pasa el resultado obtenido por una capa de pooling. Hay varios tipos de pooling pero el más común es el max-pooling que consiste en coger el valor máximo de los píxeles que se encuentren en una zona del tamaño del filtro que se elija, por ejemplo de tamaño 3×3. Al aplicar este filtro se reduce el tamaño de la imagen para que pueda ser procesado con menos coste por las capas posteriores.

Normalmente se aplica primero una capa de convolución y luego una de pooling varias veces, hasta obtener un vector de tamaño adecuado de entrada para el resto de la red neuronal. Además a la hora de entrenar, los pesos de los distintos filtros de las distintas capas de convolución se actualizan igual

que si fueran pesos de una capa normal. Esto tiene la ventaja de que no hay que definir los pesos a mano previamente, sino que es el propio proceso de entrenamiento el que se encarga de fijarlos.

Para un mayor desarrollo de este tipo de redes neuronales se puede consultar el capítulo 12 del libro de Digital Image Processing [3]

3

La librería PyTorch

En este capítulo se verá las principales funcionalidades que nos ofrece PyTorch a la hora de definir y usar redes neuronales. Al igual que en el capítulo 2 se verá cómo definir modelos lineales para posteriormente definir redes neuronales multicapa, y finalmente, redes convolucionales. Además se explicará cómo aplica esta librería el algoritmo de descenso por gradiente a la hora de entrenar las redes neuronales.

En la sección 3.1 se hará una descripción de la librería, así como los pasos para instalarla, en la 3.2 se explica cómo maneja PyTorch los datos que se usan para entrenar las redes neuronales y cómo calcula los gradientes PyTorch. En la sección 3.3 se tratará cómo definir y entrenar los distintos tipos de redes neuronales feedforward vistos en la sección 2.1. Por último la sección 3.4 se ve cómo definir y entrenar redes convolucionales.

A lo largo de este capítulo, el código usado se ha obtenido del curso de Deep learning de IBM [4]. También se ha utilizado como fuente de información la documentación de la librería que se encuentra en la página web oficial de PyTorch [5]

3.1. Introducción a la Librería PyTorch

3.1.1. Descripción

PyTorch es una librería de Python de código abierto implementada por Facebook. Esta librería tiene varios módulos interesantes para el desarrollo de redes neuronales. Gracias al uso de estos módulos, PyTorch, a diferencia de otras librerías, calcula los gradientes de los pesos de las distintas capas a la vez que aplica la fase forward a los datos de train durante el entrenamiento de la red. Este proceso se explica de manera más detallada en la sección 3.3 Esto supone una gran ventaja frente a otras librerías como Keras que necesitan calcular de manera estática los gradientes para aplicar posteriormente el algoritmo. Otra de las ventajas de esta librería es la posibilidad de crear tus propios módulos personalizados, lo cual permite al programador tener una mayor libertad a la hora de diseñar las redes neuronales, permitiendo la creación de éstas de forma más específica para resolver

problemas concretos.

3.1.2. Instalación

En esta sección se tratará como instalar PyTorch para poder trabajar correctamente con él en las secciones posteriores. Las instrucciones detalladas para instalarlo se encuentran en la web oficial de PyTorch [5], en la cual se indican los pasos para instalar PyTorch en las distintas distribuciones de Linux. Como requisitos previos a instalarla se ha de tener instalado Python así como su librería `numpy`. Con esto instalado se puede descargar la librería usando el gestor de paquetes de Anaconda con los siguientes comandos. En las siguientes secciones no se usará procesamiento paralelo (CUDA) por lo que se instalaría así:

```
conda install pytorch-cpu torchvision-cpu -c pytorch
```

También existe la opción de instalarlo usando el gestor de paquetes de Python, pip, en cuyo caso los comandos a ejecutar serán los siguientes:

```
pip3 install https://download.pytorch.org/whl/cpu/torch-1.0.1.post2-cp37-cp37m-
    linux_x86_64.whl
pip3 install torchvision
```

Una vez instalado comprobaremos que funciona correctamente haciendo por ejemplo en la consola de Python:

```
import torch
```

3.2. Manejo de datos y de gradientes en PyTorch

3.2.1. Manejo de datos

PyTorch trabaja con tensores, que son matrices multidimensionales con un mismo tipo de datos. Para construir un tensor se usa la función

```
torch.tensor(data, dtype=None, device=None, requires_grad=False)
```

que recibe como argumentos una lista o una lista de listas si la matriz tiene más de una dimensión, el tipo de datos que va a contener la matriz, para lo cual se usa el argumento `dtype` y finalmente `requires_grad` que indica si se necesita calcular un gradiente para este tensor o no. El siguiente código es un ejemplo de cómo se declararía un tensor en PyTorch:

```
import torch
my_tensor = torch.tensor([1, 2, 3, 4, 5],dtype=torch.int32,requires_grad=True)
```

Para poder entrenar las redes neuronales se necesita calcular el gradiente, para lo cual a la hora de declarar los pesos tendremos que hacerlo con el argumento de `requires_grad = True`. Como se ha mencionado antes, esto indica que se quiere calcular el gradiente de este tensor. Cómo aplica esto PyTorch a la hora de calcular los gradientes de los pesos y bias de las sucesivas capas de una red feedforward se estudiará en la sección 3.3 de este capítulo.

A continuación se hablará de las dos clases que se usan en PyTorch para el manejo de datos a la hora de entrenar nuestras redes neuronales. Se trata de la clase `torch.utils.data.Dataset` y la clase `torch.utils.data.DataLoader`. La clase `torch.utils.data.Dataset` es una clase abstracta de la que heredan las clases que se usarán como conjuntos de datos. Para esta clase hay que implementar los métodos `__len__` y el método `__getitem__`.

La clase `torch.utils.data.DataLoader` se usará para entrenar la red neuronal, y sirve para determinar el orden en el que se van a coger los datos de un determinado conjunto para entrenar la red. De todos los argumentos de la clase `torch.utils.data.DataLoader` los más importantes son `dataset` y el `batch_size`. Este último determina el número de datos que se van a procesar por la red neuronal a la vez en un batch, durante el entrenamiento. En el caso de que queramos que los datos se permuten de manera aleatoria en cada época del entrenamiento usaremos el argumento `shuffle=True`. Los argumentos de entrada `sampler` y `batch_sampler` permiten definir cómo coger los datos del dataset con una determinada estrategia. La diferencia es que el `batch_sampler` coge un batch de índices, y ambos requieren del argumento `shuffle=False`. Finalmente si el argumento es `drop_last=True`, entonces el último batch se descarta si está incompleto, es decir, si no tiene el tamaño definido en el argumento `batch_size`.

Como se verá en el capítulo 4 de este documento, vamos a trabajar con conjuntos de datos de problemas típicos de regresión y clasificación y redes convolucionales. Todas ellas serán redes con una o más capas ocultas. A continuación se muestra como se cargarían esos datos en un dataset y como se crearía el `DataLoader` correspondiente.

```
import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader

class Data(Dataset):
    def __init__(self,file,x_columns):
        dataset = np.loadtxt(file, delimiter=",")
        x = dataset[:,0:x_columns]
        y = dataset[:,x_columns]
        self.x=torch.from_numpy(x).type(torch.FloatTensor)
        self.y=torch.from_numpy(y).type(torch.FloatTensor).view(self.x.shape[0],1)
        self.len=y.shape[0]

    def __getitem__(self,index):
        return self.x[index],self.y[index]
```

```
    def __len__(self):
        return self.len

# A continuacion se usa la clase Data para cargar un dataset y crear un dataloader
dataset1 = Data("../../data/pima-indians-diabetes.csv",8)
trainLoader = DataLoader(dataset1,batch_size=10)
```

Como se puede ver en este ejemplo el conjunto de datos se cargan desde un archivo csv.

3.2.2. Gradientes en PyTorch

Para calcular gradientes en PyTorch se usan los objetos `torch.autograd.Variable` que es un wrapper de los tensores. Al constructor de esta clase se le pasa el tensor, el cual se almacenará en el atributo `data`. Además cuenta con un atributo `grad` que almacena el valor del gradiente y el atributo `grad_fn`. Este último contiene un objeto de la clase `torch.autograd.Function`. Este objeto es lo que usa PyTorch para entrenar la regresión lineal y cualquier red neuronal. Es una clase abstracta y las clases que heredan de ella han de definir las funciones `forward` y `backward`.

El siguiente código es un ejemplo de una clase que hereda de `torch.autograd.Function`:

```
import torch.autograd.Function as Function

class Exp(Function):

    @staticmethod
    def forward(ctx, i):
        result = i.exp()
        ctx.save_for_backward(result)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        result, = ctx.saved_tensors
        return grad_output * result
```

En este ejemplo ambos métodos son estáticos ya que es código propio de la librería y así se evita su modificación. Se puede ver que utiliza una variable contexto (`ctx`) donde se almacena el valor del tensor tras aplicarle la función `forward`, el cual se usa luego para calcular el gradiente. En la función backward el valor antes almacenado en la variable `ctx` se recupera y se multiplica por la salida del gradiente anterior aplicando la regla de la cadena. Este objeto `Function` lo crea PyTorch automáticamente al aplicar cualquier operación sobre un objeto `Variable`, generando un grafo en el que se guarda la relación entre entradas y salidas, de tal manera que luego se usa para calcular el gradiente.

3.3. Arquitectura de una red neuronal en PyTorch

En esta sección se mostrará cómo se construyen redes neuronales feedforward en PyTorch empezando por la más simple, sin capas ocultas y acabando con redes de varias capas ocultas. Aunque se pueden definir redes neuronales recurrentes, no se tratará su implementación en este documento.

3.3.1. Modelos Lineales

La estructura básica de una red neuronal consta de los siguientes elementos: un `init`, una función `forward`, una función `backward`, una función `train` y una función `predict`. A continuación se muestra un código de una red neuronal definida en Python sin usar las herramientas que nos proporciona PyTorch, para posteriormente, siguiendo la misma estructura, ver cómo se haría aprovechando la funcionalidad de PyTorch. El código muestra la creación de una red de regresión lineal.

```
class Neural_Network(nn.Module):
    def __init__(self,):
        super(Neural_Network, self).__init__()
        # parameters
        self.inputSize = 2
        self.outputSize = 1
        # weights
        self.W1 = torch.randn(self.inputSize, self.outputSize) # 1 X 2 tensor

    def forward(self, X):
        self.z = torch.matmul(X, self.W1) # 1 X 1
        o = self.sigmoid(self.z) # activation function
        return o

    def backward(self, X, y, o):
        self.o_error = y - o # error in output
        #la funcion torch.t intercambia las dimensiones 0 y 1 del tensor que recibe, e,
             este caso los datos de entrada a la red
        self.W1 += torch.matmul(torch.t(X), self.o_error)

    def train(self, X, y,epoch):
        # forward + backward pass for training
        for i in range(epoch):
            o = self.forward(X)
            self.backward(X, y, o)

    def predict(self,xPredicted):
        print ("Predicted data based on trained weights: ")
        print ("Input (scaled): \n" + str(xPredicted))
        print ("Output: \n" + str(self.forward(xPredicted)))
```

PyTorch tiene una serie de módulos que permiten crear redes con esta estructura sin tener que definir dicha red solo con código Python. En la siguiente sección se verá cómo definir una red como la del código anterior en PyTorch.

Regresión lineal

En el primer código se codifican todas las partes de una red neuronal de manera directa, pero PyTorch tiene una serie de funciones que simplifican como definir una red neuronal. Siguiendo la estructura definida anteriormente de `init`, `forward`, `backward`,`train` y `predict` se va a ver cómo se codificarían estas funciones en PyTorch.

La forma más simple de definir una red neuronal en PyTorch es usando la clase `torch.nn.Sequential(*args)` a la que se le pasan los distintos módulos de nuestra red neuronal, los cuales se definen con la función `torch.nn.Linear(in_features, out_features, bias=True)`. Cada capa que se crea con esta función sigue la transformación lineal:

$$\mathbf{y} = W^{n \times m}\mathbf{x} + \mathbf{b}$$

Los argumentos que se le pasan a esta función determinan el tamaño de la matriz W, siendo $n = out\_features$ filas y $m = in\_features$ columnas y $\mathbf{b}$ un tensor de tamaño $out\_features$.

El siguiente código muestra cómo se definiría una red neuronal de regresión lineal usando los módulos de PyTorch:

```
import torch
# creamos la red
model = torch.nn.Sequential(torch.nn.Linear(2,1))
```

Al inicializar dicha red neuronal mediante la clase `torch.nn.Sequential` se crean los tensores de pesos y de bias y se inicializan a valores aleatorios siguiendo una distribución uniforme entre $-1/\sqrt{len(W)}$ y $1/\sqrt{len(W)}$ por defecto. También se define la función `forward` asociada a dicha red de tal manera que se puede hacer `yhat=model.forward(x)` para la fase de predecir un valor a partir del tensor de entrada.

Normalmente en PyTorch se usa un sistema más elaborado que el anterior, definiéndonos nuestra propia clase para cada red neuronal, de tal manera que definimos todas las funciones que componen la estructura de la red neuronal:

```
from torch import nn

class linear_regression(nn.Module):

    def __init__(self, input_size, output_size):
        super(linear_regression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)
```

```
def forward(self, x):
    yhat = self.linear(x)
    return yhat

def train(self, optimizer, criterion, train_loader, epochs=100):
    for epoch in range(epochs):
        for x, y in train_loader:
            # en este caso la llamada a self sobrecarga a la funcion forward
            #por lo que en la siguiente linea es como si se llamara a la funcion
                forward de esta clase
            yhat = self(x)
            loss=criterion(yhat, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step())

def predict(self,xpredict):
    return self(xpredict)
```

La clase `linear_regression` hereda de `torch.nn.Module`, que es la clase base sobre la que se crean todas las redes neuronales en PyTorch. Sobrescribe dos funciones, la función `__init__`, en la que se recibe el tamaño de entrada de la red y el de salida, y la función `forward` que es la función que define la transformación que se aplica en la neurona, es decir es el método `nn.Linear(input_size, output_size)` de la función `__init__` descrita antes.

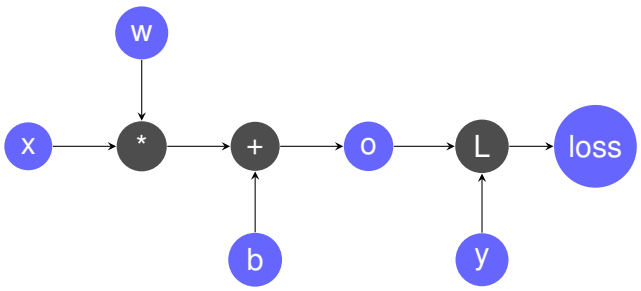
Además se define la función `predict`, que simplemente aplica la función `forward` al dato del que se quiere obtener la predicción. Como se observa en el código anterior, no se llama a la función `forward` sino que se sobrecarga la llamada a `self` de tal manera que se hace `self(x)` para llamar a la función `forward`. La función `train` es algo más compleja. Para entrenar una regresión lineal lo primero que se aplica es la función `forward` que se ha definido anteriormente, es decir se obtiene la predicción para el batch de datos (el batch que se ha definido anteriormente a la hora de declarar el `train_loader`) aplicando la fórmula del forward con los pesos y el bias que tiene definidos en cada época del entrenamiento. En segundo lugar se aplica la función de pérdida para ver cómo de lejos está la red de predecir el valor correcto de ese dato. Para el caso de la regresión lineal el argumento `criterion` tiene que definirse previamente ya que se le pasa a la función `train`, que es la función de pérdida del error cuadrático medio cuya fórmula es:

```
torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean').
```

Para entrenar la regresión lineal usaremos la función con sus valores por defecto, con los cuales calcula el error cuadrático medio que hay entre la predicción de la regresión lineal y el valor esperado de esos datos de entrenamiento.

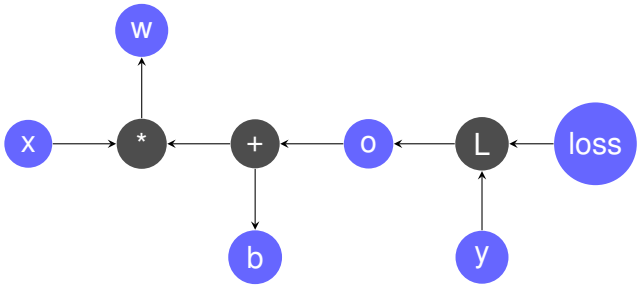
Una vez se tiene la función de error, se aplica el método de descenso por gradiente para actualizar

los pesos y el bias actuales con el objetivo de minimizar el error. Para ello se calcula el gradiente de la función de pérdida mediante la función `backward`, que se define en la propia librería PyTorch. Esta fase del entrenamiento es una de las que diferencian a PyTorch de otras librerías como Keras. PyTorch usa un sistema para calcular el gradiente que es similar al usado por la librería TensorFlow de Google, la cual también trabaja con tensores y con unos objetos llamados `Variables` que actúan como wrapper de los tensores. En el caso de la regresión lineal, al aplicar la función `forward`, se realizan una serie de operaciones sobre los tensores de pesos y sobre el bias. Estas operaciones generan el siguiente grafo:



Este grafo representa las entradas y salidas durante la fase forward del entrenamiento. Las entradas constituyen los argumentos que se le pasan a la función `forward` del objeto `Function` correspondiente. Cada nodo gris tiene asociado un objeto `Function` con sus funciones `forward` y `backward` definidas y se almacena el objeto en el atributo `grad_fn` de las variables que son sus entradas. La w (pesos) y la x (datos) son entradas de la operación * (multiplicación). La salida de la multiplicación, junto con la b (bias) son entradas a la operación + (suma). La salida de ésta es la entrada de L (la función de pérdida) junto con y (valor esperado de los datos de entrada).

El grafo que siguen las entradas y salidas de las funciones backward de los distintos objetos `Function` almacenados en el atributo `grad_fn` es el siguiente:



Los valores numéricos de los gradientes se almacenan en el atributo `grad` en los nodos w (pesos) y b (bias). Estos valores serán lo que se utilice para actualizar los pesos y el bias. Este proceso se lleva a cabo a través del optimizador, que es un objeto que se le pasa como argumento a función de `train`. Este objeto puede ser de varios tipos, pero para este ejemplo usaremos:

```
torch.optim.SGD(params, lr=<required parameter>,
                momentum=0, dampening=0, weight_decay=0,
                nesterov=False)
```

que aplica el modelo de "Standard Gradient Descent". Este objeto actualiza los valores de los pesos y el bias con el algoritmo de descenso por gradiente usando los valores almacenados en el atributo `grad`. Este paso se realiza en la linea `optimizer.step()` del código de la función de `train`. Previamente se ha de inicializar el gradiente a cero para cada batch del conjunto de datos de entrenamiento, para lo cuál se usa la función `optimizer.zero_grad()`, que pone a cero el atributo `grad` de los pesos y del bias.

Regresión logística

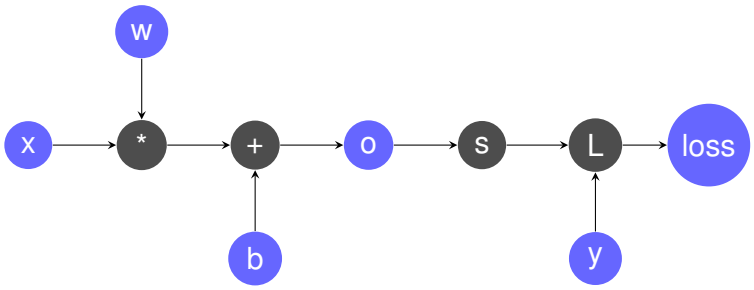
Para problemas de clasificación con dos clases se usa la regresión logística. La manera de inicializar una regresión logística es muy similar a la de inicializar la regresión lineal. Se muestra en el siguiente código:

```
from torch import nn

class logistic_regression(nn.Module):
    def __init__(self, input_size, output_size):
        super(logistic_regression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        yhat = torch.sigmoid(self.linear(x))
        return yhat

    def train(self, optimizer, criterion, train_loader, epochs=100):
        for epoch in range(epochs):
            for x, y in train_loader:
                yhat=self(x)
                loss=criterion(yhat, y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step())
    def predict(self,xpredict):
        return self(xpredict)
```

La única diferencia es que aplica la función sigmoide a la salida de la transformación lineal. Esta función transforma esta salida en un valor entre 0 y 1. Si este valor es superior al 0.5 entonces se clasifica como clase 1 y si no como clase 0. También hay diferencias a la hora de entrenarla. Se le pasa otra función de pérdida a la función `train` como argumento `criterion`, la "Binary Cross Entropy", que en PyTorch es:

```
torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')
```

El grafo que se generaría en este caso sería el siguiente:



Se añade el nodo s (función sigmoide) al grafo anterior. La fase de actualización de pesos es igual que para la regresión lineal.

3.3.2. Redes neuronales multicapa

Regresión mediante redes neuronales multicapa

Al igual que en la sección anterior la arquitectura de una red neuronal multicapa tiene la misma estructura y necesita unas funciones `init`, `forward`, `backward`, `train` y `predict`. A continuación se muestra el código para definir una red neuronal multicapa en PyTorch:

```
from torch import nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self, Layers):
        super(Net,self).__init__()
        self.hidden = nn.ModuleList()
        for input_size, output_size in zip(Layers,Layers[1:]):
            self.hidden.append(nn.Linear(input_size, output_size))

    def forward(self, activation):
        L=len(self.hidden)
        for (l,linear_transform)  in zip(range(L),self.hidden):
            if l<L-1:
                activation = F.relu(linear_transform (activation))
            else:
                activation = linear_transform (activation)
            return activation

         def train(self, optimizer, criterion, train_loader, epochs=100):
         for epoch in range(epochs):
             for x,y in train_loader:
                 yhat = self(x)
                 loss = criterion(yhat,y)
                 optimizer.zero_grad()
                 loss.backward()
                 optimizer.step())
```

```
    def predict(self,xpredict):
        return self(xpredict)
```

El código mostrado nos permite crear una red neuronal con un número variable de capas ocultas y de neuronas de cada capa. Para ello usa la función `torch.nn.ModuleList(modules=None)` que permite añadir módulos (capas de neuronas) a una lista que posteriormente se usará para clasificar los datos que entren para poder entrenar la red neuronal. La función `forward` va aplicando sucesivamente los módulos anteriores junto con la función `torch.nn.functional.relu`. Al constructor de esta clase se le pasa una lista con el tamaño de las distintas capas ocultas.

A la hora de entrenar la red neuronal se siguen los mismos pasos que para la regresión lineal de una capa: se aplica al función `forward`, y se compara el resultado obtenido con el resultado esperado a través de la función de pérdida. La función `backward` funciona igual, pero se añaden más nodos al grafo que se genera para calcular el gradiente.

Clasificación mediante redes neuronales multicapa

Para resolver problemas de clasificación complejos o con más de dos clases se usan redes neuronales multicapa. A continuación se muestra un código para definir dichas redes y para resolver problemas de clasificación.

```
from torch import nn
import torch.nn.functional as F

class Net2(nn.Module):
    def __init__(self, Layers):
        super(Net2,self).__init__()
        self.hidden = nn.ModuleList()
        for input_size,output_size in zip(Layers, Layers[1:]):
            self.hidden.append(nn.Linear(input_size, output_size))

def forward(self, activation):
    L=len(self.hidden)
    for (l,linear_transform)  in zip(range(L), self.hidden):
        if l<L-1:
            activation = F.relu(linear_transform (activation))
        else:
            activation = torch.sigmoid(linear_transform (activation))
    return activation

def train(self, optimizer, criterion, train_loader, epochs=100):
    for epoch in range(epochs):
        for x, y in train_loader:
            yhat = self(x)
            loss = criterion(yhat,y)
```

```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

def predict(self,xpredict):
    return (self(xpredict) > 0.5)
```

Se usa aquí la clase `torch.nn.ModuleList(modules=None)`. La principal diferencia es que se aplica la función `sigmoid` a la salida. La función `predict` de este código es para resolver problemas con sólo dos clases Para conjuntos de datos con tres o más clases el tensor de salida ha de tener el mismo tamaño que el número de clases. Por ejemplo, si nuestro problema tiene tres clases, la lista de tamaños que se le pasa al constructor tendrá como último valor 3. También se ha de modificar la función predict en este caso, que ahora sería la siguiente:

```
def predict(self,xpredict):
    yhat=self(xpredict)
    _,yhat=torch.max(yhat,1)
    return yhat
```

Se usa la función `torch.max` para determinar cuál es el índice en el que está el valor máximo de la salida.

Para entrenar una red con conjuntos de datos con dos clases se puede usar el criterio de "Binary Cross Entropy", al igual que en el caso de la regresión logística de la sección anterior. Sin embargo, para problemas con tres o más clases se usará `torch.nn.CrossEntropyLoss` como criterio para calcular la pérdida a la hora de entrenar la red neuronal. Al igual que en el caso de la regresión logística se usará la función sin argumentos. El resto del proceso de entrenamiento es igual que en las secciones anteriores.

3.4. Redes Convolucionales

En esta sección se verá cómo se implementan redes convolucionales en PyTorch. Como ya se habló en el capítulo 2 las redes neuronales convolucionales se usan para resolver problemas relacionados con clasificación de imágenes, para lo cual cuentan con dos tipos de capas más aparte de las propias de una red neuronal feedforward. Estas capas son una o varias capas convolucionales y una o varias capas de pooling. Al igual que en otras librerías como Keras, PyTorch cuenta con implementaciones de estas capas. En el módulo `torch.nn` contiene una serie de funciones que nos permiten definir las distintas capas convolucionales de nuestra red neuronal. Estas funciones nos permiten aplicar convoluciones de 1, 2 y 3 dimensiones y aplicar la convolución transpuesta también. En el caso de las redes neuronales no se suelen utilizar las convoluciones transpuestas. Todas las funciones tienen una cabecera similar; por ejemplo la función:

```
torch.nn.Conv2d(in_channels, out_channels,
                kernel_size, stride=1, padding=0,
                dilation=1, groups=1, bias=True)
```

nos permite definir una capa convolucional que aplica una convolución en dos dimensiones. Entre sus argumentos se encuentran `in_channels` y `out_channels` que son el número de canales de entrada y de salida de la capa convolucional. El argumento `kernel_size` nos permite definir el tamaño del kernel a usar para hacer la convolución. De los otros argumentos son interesantes en primer lugar el `bias` que se usa par indicar si se va a usar éste para computar la convolución y el `padding` que se usa para determinar si se va a usar padding a la hora de calcular la convolución. Para la capa de pooling también utilizaremos el módulo `torch.nn`, que tiene funciones que nos permiten crear estas capas. Estas funciones nos permiten aplicar maxpooling en varias dimensiones. Al igual que antes, las cabeceras de las funciones de maxpooling son iguales para todos los tipos distintos. Un ejemplo es la cabecera del caso de maxpooling en 2 dimensiones que es:

```
torch.nn.MaxPool2d(kernel_size, stride=None,
                   padding=0, dilation=1,
                   return_indices=False, ceil_mode=False)
```

Como argumentos recibe el tamaño del kernel que se va a usar para calcular el pooling, del cual dependerá el tamaño de la imagen de salida. Al igual que antes podemos indicar si queremos que se aplique padding al aplicar el kernel con el argumento `padding`. El resto de argumentos son `dilation` que se usa para determinar cuánto se va a desplazar el kernel después de aplicarlo a determinado pixel y `ceil_mode=True` que hace que se use la función $ceil$ en vez de la función $floor$ para aplicar el filtro.

Los pesos de la capa convolucional, al igual que el resto de los pesos de la red neuronal, se actualizan en la fase de entrenamiento de la red, haciendo que no sea necesario definir los filtros, ya que es la propia red la que se encarga de determinar cuáles son los más adecuados para cada problema ya que estos pesos también se actualizan durante la fase de entrenamiento.

A continuación se muestra cómo se definiría una red neuronal convolucional en PyTorch:

```
from torch import nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self,out_1=2,out_2=1):
        super(CNN,self).__init__()
        self.cnn1=nn.Conv2d(in_channels=1,out_channels=out_1,kernel_size=2,padding=0)
        self.relu1=nn.ReLU()
        self.maxpool1=nn.MaxPool2d(kernel_size=2 ,stride=1)
        self.cnn2=nn.Conv2d(in_channels=out_1,out_channels=out_2,kernel_size=2,stride
            =1,padding=0)
        self.relu2=nn.ReLU()
```

```
        self.maxpool2=nn.MaxPool2d(kernel_size=2 ,stride=1)
        self.fc1=nn.Linear(out_2*7*7,2)

    def forward(self,x):
        out=self.cnn1(x)
        out=self.relu1(out)
        out=self.maxpool1(out)
        out=self.cnn2(out)
        out=self.relu2(out)
        out=self.maxpool2(out)
        out=out.view(out.size(0),-1)
        out=self.fc1(out)
        return out
    def train(self, optimizer, criterion, train_loader, epochs=100):
        for epoch in range(epochs):
            for x,y in train_loader:
                yhat=self(x)
                loss=criterion(yhat,y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step())

    def predict(self,xpredict):
        return (self(xpredict))
```

4

Ejemplos de uso de PyTorch

En este capítulo se resolverán algunos problemas típicos de redes neuronales usando las funciones de la librería PyTorch que se han explicado en el capítulo 3. Se resolverán tres problemas: la clasificación de pacientes según la predicción de tener diabetes (clasificación), la predicción del precio de una vivienda (regresión) y por último un clasificador de números escritos a mano. En la sección 4.1 se verá cómo resolver el problema de predicción del precio de viviendas. Las otras secciones tratan de problemas de clasificación. En la sección 4.2 se verá la detección de pacientes con diabetes y en la sección 4.3 se resolverá un problema de clasificación más complejo, el de clasificación de números manuscritos. Por último, usando el problema de la sección 4.3, en la sección 4.4 se comparan los tiempos de entrenamiento de la librerías PyTorch y Keras.

4.1. Predicción del precio de viviendas

4.1.1. Descripción del problema y de los datos

Este problema es típico de regresión lineal. Consiste en predecir el precio medio de viviendas de 506 zonas de Boston en función de los siguientes datos:

- CRIM: ratio de crimen per capita por población
- ZN: proporción de áreas residenciales zonificadas con más de 25000 pies cuadrados.
- INDUS: proporción de acres no comerciales por ciudad
- CHAS: Charles River dummy variable (1 si el tramo se encuentra en el río, 0 en caso contrario).
- NOX: concentración de óxidos nítricos.
- RM: número medio de habitaciones por vivienda.
- AGE: proporción de viviendas ocupadas construidas antes de 1940.
- DIS: distancias ponderadas a cinco centros de empleo.
- RAD: índice de accesibilidad a las autopistas radiales.
- TAX: ratio de impuestos a la propiedad por cada 10.000$.
- PTRATIO: ratio de alumnos-profesores.
- B: $1000(Bk - 0,63)2$ donde Bk es la proporción de ciudadanos negros por población.

- LSTAT: porcentaje de la población de clase baja.

En este caso los datos de este problema se han obtenido a partir de un csv descargado de la página web de Kaggle y consta de 506 filas y 14 columnas donde la última es el precio de la vivienda en miles de dólares.

Lo primero que se ha de hacer para resolver este problema es cargar los datos que se usarán para entrenar y evaluar la red neuronal que definiremos para resolverlo. Los datos han de cargarse en un dataset, que es la clase que utiliza PyTorch para trabajar. Y con este dataset se crearán posteriormente los dataloaders de train y de test que se usarán para entrenar la red neuronal. Para automatizar el proceso de cargar los datos a partir de un csv creamos la clase `Data` que crea un objeto `Dataset`, cuyo constructor recibe los siguientes parámetros:

```
dataHousing = Data("../../data/boston_housing.csv",0,13)
```

El primero es la dirección donde se encuentra el archivo csv, el segundo es la columna donde empiezan los datos, y el último indica cuál es la última columna, donde se encuentran los valores numéricos que se corresponden con el precio que se predice para las viviendas en base a los datos. El código completo de esta clase se encuentra en el apéndice A.1 de este documento.

Para conseguir unos mejores resultados se normalizarán los datos, para lo cual se usa `sklearn.preprocesing.StandardScaler` de la librería `sklearn` de python, cuya documentación se encuentra en su página web [6]. Esto se hace dentro de una función de la clase `Data` que es:

```
    def normalizar(self):
        std_sc = StandardScaler()
        self.x = torch.from_numpy(
                 std_sc.fit_transform(self.x.numpy())).type(torch.FloatTensor)
        return

    dataHousing.normalizar()|.
```

Una vez hecho esto fijaremos la semilla de `torch` para que las particiones sean siempre las mismas de manera que podamos comparar los dos modelos a utilizar. Esto se hace mediante la siguiente función: `torch.manual_seed(1)`.

4.1.2. Definición y entrenamiento del modelo

A continuación definiremos el modelo, para lo cual usaremos la clase del capítulo 3 que nos permitía definir una red neuronal multicapa para problemas de regresión con alguna modificación que nos permite definir cómo se inicializan los pesos de la red. Para añadir esta modificación se le ha de pasar un argumento más a la función `__init__`, en el que se indica cómo se van a inicializar los pesos. Vamos a trabajar sólo con dos formas de inicializar los pesos, una que los inicializa según una distribución

uniforme y otra que aplica la inicialización xavier. A continuación se pueden ver las funciones que se usan para estas formas de inicializar los pesos:

```
#inicializa segun una uniforme
linear.weight.data.uniform_(0, 1)
#inicializacion xavier
torch.nn.init.xavier_uniform_(linear.weight)
```

Una vez definido el tipo de inicialización que queremos, procedemos a construir la red neuronal con la que se resolverá el problema predicción del precio de viviendas. El siguiente código hace uso de la clase `RedNeuronal` que es la modificación de la clase definida en el capítulo 3 para una red neuronal multicapa. Para este problema se van a utilizar dos aproximaciones, un modelo lineal y una red neuronal multicapa. El primero de ellos se define con el siguiente código:

```
LAYERS=[13,1]
ActivationFunctions = []
model = RedNeuronal(LAYERS,ActivationFunctions,"xavier")
```

En este caso no se aplica ninguna función de activación a la salida de la red ya que es un problema de regresión lineal. Se inicializan los pesos con la inicialización de xavier como indica el último parámetro.

También se usará otra aproximación que en este caso será una red multicapa cuyo código es el siguiente:

```
LAYERS=[13,20,1]
ActivationFunctions = [F.relu]
model = RedNeuronal(LAYERS,ActivationFunctions,"uniform")
```

Esta red neuronal tiene una capa oculta, de 20 neuronas con entrada de tamaño 13 y salida de tamaño 20 , y la capa de salida con una entrada de 20 y salida de 1. Se aplica una función ReLU a la salida de la capa oculta.

Para los dos modelos definidos se usa la función train de la clase `RedNeuronal`. Se usa el método de descenso por gradiente para entrenar la red y la función de pérdida de "mean square error". El código es el siguiente:

```
learning_rate = 0.01
criterion=torch.nn.MSELoss()
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate,weight_decay=0.01)
LOSS=[]
trainLoader=DataLoader(dataset=dataTrain,batch_size=5)
LOSS,epoch,min_loss=model.train(dataHousing,optimizer,criterion,trainLoader,100)
```

Cabe destacar el argumento `weight_decay` del `optimizer`, que se usa para aplicar regularización, concepto que se vio en el capítulo 2. El proceso de elegir un peso adecuado para aplicar regularización a la hora de entrenar la red neuronal es complejo puesto que normalmente se crea lo que se denomina un grid de valores donde se prueban varias combinaciones de pesos de regularización y de tasas de

aprendizaje para ver cual obtiene mejores resultados. Para nuestro problema este proceso se llevó a cabo de manera manual obteniendo los valores que aparecen en el código anterior. Los valores que se probaron fueron $[10^-3, 10^-2, 10^-1, 1, 10]$ tanto para los pesos de regularización como para la tasa de aprendizaje, lo cual se evaluó usando Validación Cruzada con 11 particiones de 46 datos cada una.

Con los valores que mejores resultados dieron se entrenó y se evaluó cada modelo usando Validación Cruzada. Para ello se usa el siguiente código:

```
num_partVC = 11
lenVC = int(len(dataHousing)/num_partVC)
lengths = ()
for i in range(num_partVC):
    lengths = lengths + (lenVC,)
#usamos validacion cruzada para validar los datos
particiones = random_split(dataHousing, lengths)
```

La función `random_split` es una función de `torch` y se le pasa como argumento el `Dataset` y una lista con las longitudes de los conjuntos en los que va a dividir los datos y devuelve una lista de datasets con la longitud especificada. Se divide el dataset con los datos de las viviendas de Boston en 11 subconjuntos de 46 datos cada uno. Para escoger qué partición se usará para los datos de test y cuales para los de train se añade una función a la clase `Data`, cuyo código es el siguiente:

```
def validacionCruzada(self, indice, particiones):
    aux = []
    for i in range(len(particiones)):
        if i != indice:
            aux = aux + [particiones[i]]
        dataTrain = torch.utils.data.ConcatDataset(aux)
        dataTest = particiones[indice]
    return dataTrain,dataTest
```

En esta función se cogen 10 de las 11 particiones para los datos de train y 1 para los datos de test. En cada iteración del bucle de validación cruzada se cambia cuál de las particiones se usa como datos de test.

La función `train` de la clase `RedNeuronal` también se modifica ya que se usa una técnica para guardar el modelo que menor pérdida ha tenido de todas las épocas para asegurarnos de que no se entrena de más el modelo. Esto se hace añadiendo el siguiente código a la función de `train`:

```
if loss.item() < min_loss:
    value = epoch
    min_loss = loss.item()
    torch.save(self.state_dict(), 'best_model.pt')
```

Para ello se almacenan los valores de los pesos y de los bias de cada capa del modelo. PyTorch almacena estos datos en un diccionario como un atributo y al que se accede mediante la función `state_dict()`. Este modelo que tiene la menor pérdida se guarda mediante la función `torch.save`

sobrescribiendo el anterior modelo con menor pérdida. A la hora de evaluarlo se carga el mejor usando:

```
model_best = RedNeuronal(LAYERS,ActivationFunctions,"Normal")
model_best.load_state_dict(torch.load('best_model.pt'))
```

4.1.3. Resultados

Se usarán dos criterios para evaluar cómo de bien resuelve el problema cada una de las dos redes neuronales que se han definido. El primero es la pérdida media que tiene nuestra predicción frente a los valores reales de los precios de esas viviendas Esta perdida media se calcula con la siguiente fórmula:

$$loss = \frac{\sum_{i=1}^{11} l_i}{11}$$

donde cada l_i es la pérdida del mejor modelo, es decir, el modelo que obtuvo la pérdida media más baja durante el entrenamiento con los datos de train de cada partición.

Además de este criterio se mirará el R2 de los datos frente a sus valores esperados, lo cual nos da una idea de cómo de bien interpola nuestra red los valores reales. A la hora de predecir los datos se aplica un filtro a los datos de salida para eliminar aquellas predicciones menores que el valor mínimo de los datos reales.

Método usado	Pérdida Media	R2
Regresión lineal	23.04	0.72
Red Multicapa	13.34	0.83

Tabla 4.1: Resultados Boston Housing

La regresión lineal obtiene una pérdida media de 23.04, lo cuál se observa en la tabla 4.1. En la imagen 4.1 se observa la relación entre la predicción de este modelo y los valores reales de la predicción. Esto se hace juntando las predicciones sobre cada partición de la validación cruzada, donde la red se ha entrenado con las otras 11 particiones restantes.

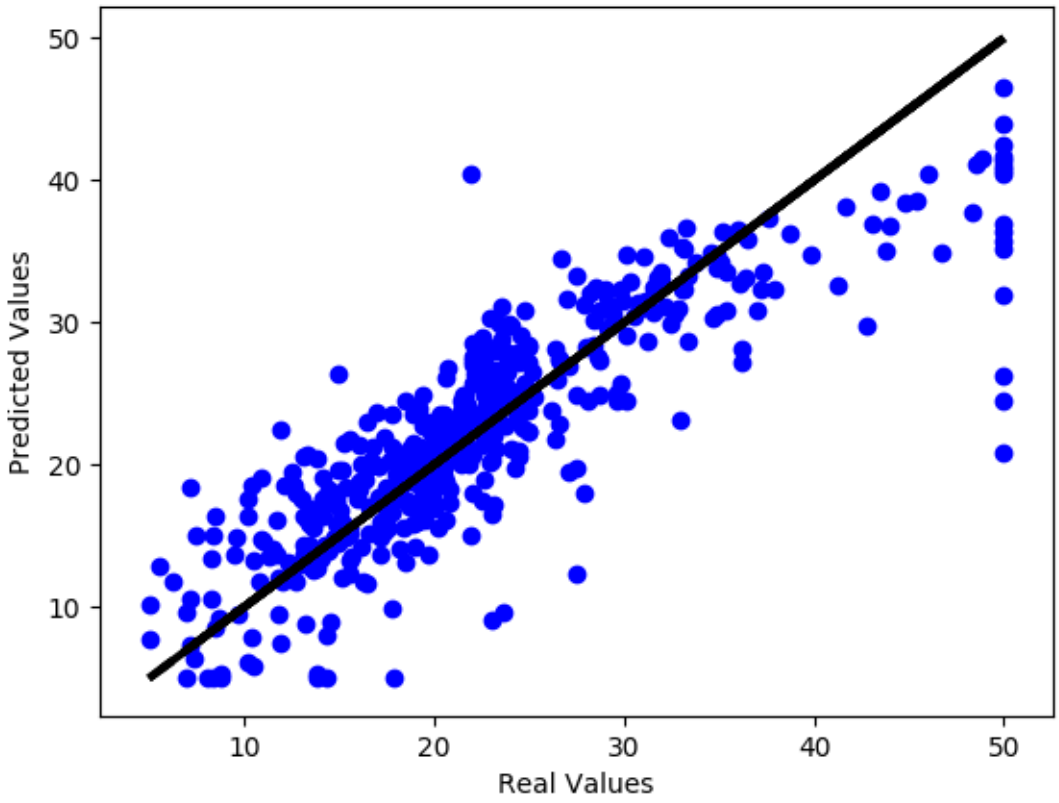


Figura 4.1: Imagen correlación entre los valores reales y los obtenidos con la regresión lineal

Como se puede ver en la tabla 4.1 los resultados de la red neuronal con varias capas son mejores, ya que se reduce el error medio y aumenta el coeficiente de correlación. En este caso se tiene una pérdida media de 13.34. Al igual que antes se muestra la relación entre los datos que predice la red neuronal con una capa oculta y los valores reales de los datos.

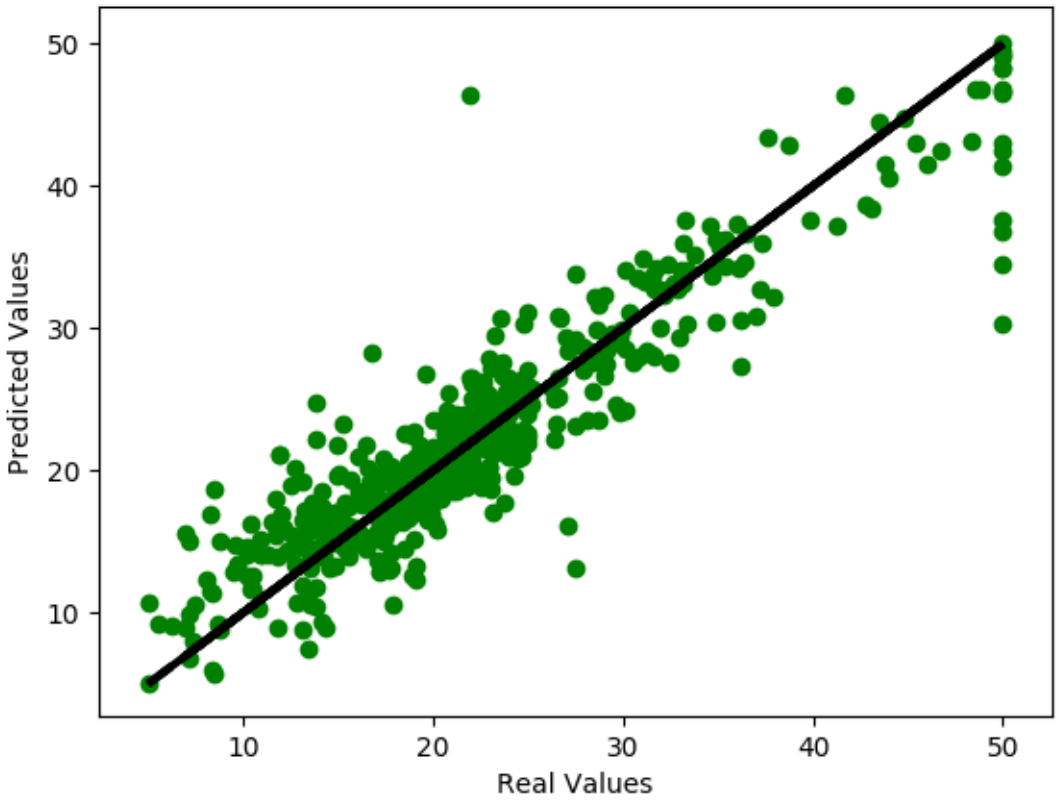


Figura 4.2: Imagen correlación entre los valores reales y los obtenidos con la red neuronal multicapa

Se observa que en este caso los datos cercanos a la parte derecha de la imagen 4.2 se predicen mejor que en el caso de la imagen 4.1.

4.2. Detección de diabetes

4.2.1. Descripción del problema y de los datos

Este problema consiste en determinar si un paciente padece o no diabetes a partir de los datos recogidos de pacientes indios y si tuvieron o no diabetes en los siguientes 5 años. Dichos datos contienen las columnas que se muestran a continuación:

- Número de embarazos.
- Concentración de glucosa.
- Presión en sangre.
- Tamaño de la doblez de la piel en el triceps.
- Insulina
- Índice de masa corporal
- Función que representa el número de antepasados con diabetes.
- Edad

Los datos se han obtenido de la página de Kaggle en formato csv y tienen 9 columnas y 768 filas.
Al igual que en problema anterior se usa la clase `Data` para cargar los datos y se normalizan con la función `normalizar` de dicha clase:

```
dataDiabetes = Data("../../data/pima-indians-diabetes.csv",0,8)
dataDiabetes.normalizar()
```

La última columna en este problema contiene las clases que se corresponden a los distintos datos de entrada. En este problema también se fijará la semilla al mismo valor que el problema anterior.

4.2.2. Definición y entrenamiento del modelo

Al igual que para el ejemplo anterior usaremos la versión modificada de la clase `RedNeuronal`. El código para definir la red para este problema será:

```
LAYERS=[8,12,8,1]
ActivationFunctions = [F.relu,F.relu,torch.sigmoid]
model = RedNeuronalMulticapa(LAYERS,ActivationFunctions,"xavier")
```

Se trata de una red neuronal de varias capas; la primera es una capa con 12 neuronas que recibe una entrada de tamaño 8 y tiene una salida de tamaño 12, la segunda una capa con 8 neuronas que recibe una entrada de tamaño 12 y tiene una salida de tamaño 8 y finalmente la capa de salida con una entrada de tamaño 8 y una salida de tamaño 1. A la salida de la primera capa y de la segunda capa se le aplica la función `F.relu` y a la salida la función `torch.sigmoid`.
Un vez definida la red neuronal procedemos a entrenarla. Para ello llamamos a la función `train` de la clase `RedNeuronal`. El siguiente código muestra cómo se llamaría esta función:

```
criterion=torch.nn.BCELoss()
learning_rate=0.01
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.01)
LOSS=[]
trainLoader = DataLoader(dataset=dataTrain,batch_size=10)
LOSS,epoch,min_loss = model.train(dataDiabetes, optimizer, criterion, trainLoader, 200)
```

Se usa como función de pérdida la 'Binary Cross Entropy' y el método de descenso por gradiente para entrenar la red neuronal. Se crearán 11 particiones para entrenar y validar el modelo del mismo modo que se crearon para el problema anterior. Para este problema se usa un valor de alpha para la regularización de 0.01 obtenido con el mismo sistema que para el problema anterior.

4.2.3. Resultados

Para este problema el criterio que se usará para determinar cómo de bien clasifica nuestra red neuronal es la media del porcentaje de aciertos sobre le conjunto de test correspondiente en el proceso de validación cruzada. Con esto en mente se procede a entrenar y evaluar el modelo mediante validación cruzada y se obtiene un porcentaje medio de acierto del 75.79 %.

Para obtener mayor información se construye la matriz de confusión sumando las matrices de confusión de cada una de las particiones de la validación cruzada, que es la matriz de la figura 4.3. También se calcula la matriz de confusión de los porcentajes medios, que es la figura 4.4

$$\begin{pmatrix} 420 & 72 \\ 111 & 153 \end{pmatrix}$$

Figura 4.3: Matriz de confusión del problema sobre la diabetes

$$\begin{pmatrix} 54,69 & 9{,}38 \\ 14{,}45 & 19{,}92 \end{pmatrix}$$

Figura 4.4: Matriz de confusión con la media de porcentaje de acierto del problema sobre la diabetes

De la matriz 4.4 se deduce que del total del porcentaje de acierto medio se tiene que el 54 % de los aciertos son verdaderos negativos, el 19 % restante son verdaderos positivos. Lo más relevante es ver que ha habido más falsos positivos que falsos negativos entre los casos de error, ya que son más graves los falsos negativos puesto que suponen que no se detecte la enfermedad.

4.3. Clasificador de números manuscritos

El último problema que se resolverá es un problema de clasificación más complejo que el anterior, el cual consiste en la clasificación de números manuscritos. Como los datos de este problema son imágenes se usará una red convolucional para resolverlo. La idea de resolver este problema es en primer lugar tener un ejemplo práctico de un problema resuelto con una red convolucional y en segundo lugar tener un problema con un volumen de datos lo suficientemente grande para poder medir los tiempos de entrenamiento con la librería PyTorch y la librería Keras y compararlos. Para resolverlo se definen dos ficheros, uno con la clase que define la red convolucional y el otro con el programa principal.

4.3.1. Descripción del problema y de los datos

Este problema consiste en clasificar una serie de números escritos a mano en 9 clases que se corresponden con los números del 0 al 9. Los datos de entrada de este problema son imágenes, las cuales pueden ser vistas como matrices de datos y se trata de un problema multiclase, es decir, que las clases no son sólo 0 o 1.

Para cargar los datos de este último problema no se usará la clase data como en los casos anteriores sino que se usará el conjunto de datasets que posee la propia librería de PyTorch. Estos datasets se encuentran en el módulo `torchvision.datasets` en la clase

```
torchvision.datasets.MNIST(root, train=True, transform=None,
                           target_transform=None, download=False)
```

que contiene este dataset entre otros. El argumento `root` indica el directorio donde se quieren descargar los datos, el argumento `train` indica si se quiere descargar el conjunto de datos de entrenamiento o el de validación y el argumento `download` indica si se quiere descargar los datos para futuras ejecuciones o sólo cargar el dataset. Por último el argumento `transform` que aplica una transformación a los datos de entrada que en nuestro caso será la siguiente:

```
IMAGE_SIZE = 28
composed = transforms.Compose([transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
                               transforms.ToTensor()])

train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
                             download=True, transform=composed)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
                            download=True, transform=composed)
```

Las posibles transformaciones se encuentran en el módulo `torchvision.transforms`. En este caso se usan dos de estas transformaciones, la transformación `transforms.Resize` que convierte las imágenes al tamaño especificado por argumento, en este caso 28×28, la transformación `transforms.ToTensor` que convierte los datos a tensores para poder trabajar con ellos en PyTorch. Estas dos transformaciones se componen para crear una sola que aplique las dos con la función `transforms.Compose`.
En el último código mostrado se ve como se obtienen los datos de train y de test. Para este ejemplo se usarán 60000 datos de entrenamiento y 10000 datos de test.

4.3.2. Definición y entrenamiento del modelo

Para resolver este problema se usa una red neuronal que se ha obtenido del curso de Deep Learning ofrecido por IBM en PyTorch [4], y al igual que para los problemas anteriores lo haremos definiendo una clase. La clase en este caso es la siguiente:

```
import torch
from torch import nn
import numpy as np
import math



class CNN(nn.Module):

    # Contructor
    def __init__(self, out_1=16, out_2=32):
        super(CNN, self).__init__()
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=out_1, kernel_size=5, padding
            =2)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)
        self.cnn2 = nn.Conv2d(in_channels=out_1, out_channels=out_2, kernel_size=5,
            stride=1, padding=2)
```

```
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        self.fc1 = nn.Linear(out_2 * 7 * 7, 10)

    # Prediction
    def forward(self, x):
        out = self.cnn1(x)
        out = self.relu1(out)
        out = self.maxpool1(out)
        out = self.cnn2(out)
        out = self.relu2(out)
        out = self.maxpool2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        return out

    def train(self, dataTrain, optimizer, criterion, train_loader, n_epochs=3):
        for epoch in range(n_epochs):
            for x, y in train_loader:
                yhat = self(x)
                loss = criterion(yhat, y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
        return

    def predict(self,dataTest):
        return self(dataTest)
```

Esta clase define una red convolucional en la que se aplica dos veces la siguiente secuencia de funciones, primero una capa convolucional, a cuya salida se aplica la función de activación `ReLU` para eliminar los resultados negativos y finalmente la capa de pooling, en este caso maxpooling. A la salida de esta red se conecta una red neuronal feedforward de una capa con entrada de tamaño $32 \times 7 \times 7$ ya que la salida de la última capa convolucional son 32 matrices de tamaño 7×7. Esto se debe a que la primera capa de pooling reduce a una matriz de 14×14 y después de la segunda a una de tamaño 7×7. En este caso las capas de convolución no reducen el tamaño de la matriz ya que aplican padding.
Una vez definida la red se procederá a entrenarla con los datos de train, para lo cual, como en todas la redes neuronales descritas hasta ahora, se han de definir los parámetros que se le pasarán a la función train. El siguiente código muestra que parámetros se han elegido para este caso y la llamada a la función train:

```
model = CNN(out_1=16, out_2=32)
criterion = torch.nn.CrossEntropyLoss()
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=100)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=5000)
```

```
model.train(train_dataset,optimizer, criterion, train_loader, 3)
```

Como función de pérdida se usará la de 'Cross Entropy Loss' ya que este caso es un problema de clasificación multiclase y como optimizador se usará el 'Standard Gradient Descent'. Se definen los `DataLoaders` de train y de test y se entrena el modelo , sólo con 3 épocas debido a que si no tardaría mucho tiempo.

4.3.3. Resultados en Test

Una vez entrenada la red convolucional se procede a comprobar su precisión con los datos de test, para ello se hace una predicción sobre los datos de test y se comprueba en cuántos ha acertado. El código es el siguiente:

```
for x_test, y_test in test_loader:
    z = model.predict(x_test)
    _, yhat = torch.max(z.data, 1)
    correct += (yhat == y_test).sum().item()
accuracy = correct / len(test_dataset)
```

En el dataset de test que se usa en este ejemplo se obtiene un 97,43 % de acierto.
Al igual que para el ejemplo de la sección 4.2 calculamos la matriz de confusión, que en este caso tiene 10 filas y 10 columnas ya que este problema tiene 10 posibles clases.

$$\begin{pmatrix} 517 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 564 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 500 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 507 & 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 482 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 434 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 495 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 0 & 0 & 0 & 510 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 482 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 487 \end{pmatrix}$$

Figura 4.5: Matriz de confusión MNIST

Se observa que los principales errores se han producido entre la clase 7 y la clase 2, ya que son números que escritos a mano son muy parecidos.

4.4. Comparación de tiempos con la librería Keras

Para este último apartado se ha consultado la documentación de Keras [8] A continuación se muestra el resultado más interesante de este problema, que es la comparación de tiempos entre Keras y PyTorch. Para esta comparación se ha puesto a entrenar una red neuronal con las mismas condiciones

en Keras y la de PyTorch y se han medido los tiempos que tarda cada una en entrenar la red con los mismos datos de entrada. Esta red es la misma que se definió en las secciones anteriores, con dos capas convolucionales y dos capas de pooling y una entrada de imágenes de tamaño 28×28 píxeles. El código de esta red neuronal definida en Keras se encuentra en el apéndice de este documento y y se ha obtenido del TFG "Herramientas modernas en redes neuronales: la librería Keras" [7].

Para comparar los resultados se usa la función `time.time()` de la librería time, se calcula el tiempo que tarda la red en entrenar con 50 épocas tanto en Keras como en PyTorch. Se calcula el tiempo 5 veces por cada una de las librerías. A continuación se presentan los mejores tiempos de esas 5 mediciones:

Librería	Tiempo(en min)
PyTorch	17
Keras	13.04

Tabla 4.2: Tabla con los mejores tiempos de entrenamiento

Se observa en la tabla 4.2 que la librería Keras tarda menos tiempo en entrenar en las mismas condiciones, esto puede deberse a que Keras tiene una fase de compilación que calcula los gradientes.

5 Conclusiones y Trabajos futuros

5.1. Conclusiones

En este último capítulo se recogerán las principales conclusiones que se han obtenido del análisis y el uso de la librería PyTorch así como una revisión de si se han cumplido los objetivos presentados en la introducción.

El primero de estos objetivos era ver la teoría matemática detrás de las redes neuronales, el cuál se ha cumplido en el capítulo 2 de este documento. El segundo objetivo era ver la estructura y funcionalidades que ofrece la librería PyTorch, el cual se ha cumplido en el capítulo 3. Por último en el capítulo 4 se ha cumplido el objetivo de resolver algunos problemas típicos usando la librería PyTorch. En la sección 4.1 se ha resuelto el problema de Boston Housing mediante una regresión lineal y mediante una red neuronal, siendo mejor el modelo de la red neuronal ya que obtiene una menor pérdida media y un mayor coeficiente R2. En cuanto a problemas de clasificación se ha resuelto un problema sencillo en la sección 4.2, que es el conocido problema de pima que permite clasificar pacientes con o sin diabetes. Finalmente en la sección 4.3 se estudia el problema de clasificación de números manuscritos, usando redes convolucionales. En la sección 4.4 se ha usado este último problema para comparar el tiempo que tarda en entrenar la librería PyTorch y compararlo con la librería Keras, ya que tiene un volumen significativo de datos. De este experimento se concluye que la librería Keras tarda menos en entrenar la red neuronal.

5.2. Trabajos futuros

En cuanto a líneas futuras de desarrollo de este proyecto se podría llevar a cabo un estudio acerca de cómo usar la hiperparametrización en PyTorch, usando por ejemplo un wrapper como Scikit-learn. También se puede estudiar cómo definir otro tipo de redes neuronales más avanzadas que no se han definido en este documento como son las redes recurrentes. Para terminar la librería PyTorch puede usar la GPU en lugar de la CPU para entrenar las redes neuronales, por lo que sería interesante ver la capacidad de esta librería para entrenar usando la GPU y compararla con otras librerías como se

ha hecho en el capítulo 4. Esto se debe a que el uso de GPU es bastante común para entrenar redes neuronales ya que permite el cálculo de productos de vectores de manera simultánea, reduciendo el coste de entrenar la red de un coste cuadrático a uno lineal.

Bibliografía

[1] E. N. S. Camperos and A. Y. A. García, *Redes neuronales: conceptos fundamentales y aplicaciones a control automático*. Pearson Educación, 2006.

[2] L. Fausett, *Fundamentals of neural networks: architectures, algorithms, and applications*. Prentice-Hall, Inc., 1994.

[3] R. C. Gonzalez and P. Wintz, "Digital image processing(book)," *Reading, Mass., Addison-Wesley Publishing Co., Inc.(Applied Mathematics and Computation*, no. 13, p. 451, 1977.

[4] Curso deep learning with python and pytorch(ibm):
url: `https://towardsdatascience.com/getting-started-with-pytorch-part-1-under`
fecha primer acceso: Noviembre, 2018

[5] Web oficial de la librería pytorch:
url: `https://pytorch.org/`
fecha primer acceso: Enero, 2019

[6] Web oficial scklearn:
url: `http://scikit-learn.org/stable/documentation.html`
fecha primer acceso: Marzo, 2019

[7] Carlos Antona Cortés, "Herramientas modernas en redes neuronales: la librería Keras".

[8] F. Chollet, "Keras: Deep learning library for theano and tensorflow," 2015.

Apéndices

A

Código fuente

A.1. Código clase Data y clase RedNeuronal

A.1.1. clase Data

```
import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import StandardScaler

class Data(Dataset):
    def __init__(self, file, first_col, x_columns):
        dataset = np.loadtxt(file, delimiter=",")
        x = dataset[:,first_col:x_columns]
        y = dataset[:,x_columns]
        self.x=torch.from_numpy(x).type(torch.FloatTensor)
        self.y=torch.from_numpy(y).type(torch.FloatTensor).view(self.x.shape[0],1)
        self.len=y.shape[0]

    def __getitem__(self, index):
        return self.x[index],self.y[index]

    def __len__(self):
        return self.len

    def normalizar(self):
        std_sc = StandardScaler()
        self.x = torch.from_numpy(std_sc.fit_transform( self.x.numpy())).type(torch.
            FloatTensor)
        return

    def validacionCruzada(self, indice, particiones):
        aux = []
        for i in range(len(particiones)):
            if i != indice:
                aux = aux + [particiones[i]]
```

```
            dataTrain = torch.utils.data.ConcatDataset(aux)
            dataTest = particiones[indice]
        return dataTrain,dataTest
```

A.1.2. clase RedNeuronal

```
import torch
from torch import nn
import numpy as np
import math
from sklearn.metrics import confusion_matrix

class RedNeuronal(nn.Module):

    #se le pasa una lista con las neuronas que contiene cada capa, el primer valor es
        el tamano del vector de entrada
    #y el utimo el valor de la salida.
    def __init__(self, Layers, ActivationFunct, typeWeights  = None ):
        super(RedNeuronal,self).__init__()
        self.hidden = nn.ModuleList()
        for input_size,output_size in zip(Layers,Layers[1:]):
            linear = nn.Linear(input_size,output_size)
            if typeWeights == "xavier":
                torch.nn.init.xavier_uniform_(linear.weight)
            if typeWeights == "uniform":
                linear.weight.data.uniform_(0, 1)
            self.hidden.append(linear)
        self.ActivationF = ActivationFunct

    #aplica la red neuronal a unos datos(activation)
    def forward(self, activation):
        L=len(self.hidden)
        for (l,linear_transform)  in zip(range(L),self.hidden):
            if l<L-1:
                activation =self.ActivationF[l](linear_transform (activation))
            else:
                if len(self.ActivationF) == len(self.hidden):
                    activation = self.ActivationF[l](linear_transform (activation))
                else:
                    activation =linear_transform (activation)
        return activation

    #entrenamiento de la red neuronal de varias capas
    def train(self, dataTrain, optimizer, criterion, train_loader, epochs=100):
        LOSS=[]
        min_loss = 10000000000
        for epoch in range(epochs):
            for x,y in train_loader:
```

```
                yhat=self(x)
                loss=criterion(yhat,y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                if loss.item() < min_loss:
                    value = epoch
                    min_loss = loss.item()
                    torch.save(self.state_dict(), 'best_model.pt')
                LOSS.append(loss.item())
        return LOSS,value,min_loss

    def predict(self, dataTest, criterion):
        yhat = self(dataTest[0:-1][0]).detach().numpy()
        yhat = np.clip(yhat,5,yhat.max())
        yhat = torch.from_numpy(yhat)
        loss = criterion(yhat,dataTest[0:-1][1])
        return loss.item()

    def scoreSigmoid(self, dataTest):
        yhat = self(dataTest[0:-1][0])
        lable = yhat > 0.5
        cm = confusion_matrix(np.array(dataTest[0:-1][1]),lable)
        return torch.mean((lable== dataTest[0:-1][1].type(torch.ByteTensor)).type(torch
            .float)),cm
```

A.2. Código problema regresión lineal

```
from Data import Data
from RedNeuronal import RedNeuronal
import torch
import torch.nn.functional as F
import numpy as np
from torch.utils.data import DataLoader,random_split
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score

num_partVC = 11
#cargo los datos de con los que voy a trabajar
dataHousing = Data("../../data/boston_housing.csv",0,13)
dataHousing.normalizar()

print(dataHousing.x[0])

torch.manual_seed(1)
#defino el criterio que se va a usar(Mean Square Error) y el optimizador(descenso por
    gradientte)
learning_rate=0.01
```

```
lenVC = int(len(dataHousing)/num_partVC)
print(lenVC)
lengths = ()
for i in range(num_partVC):
    lengths = lengths + (lenVC,)
#usamos validacion cruzada para validar los datos
particiones = random_split(dataHousing, lengths)

#red de una sola capa
LAYERS=[13,1]
ActivationFunctions = []
ACC=[]
r2=[]

criterion=torch.nn.MSELoss()
print("modelo lineal")
fig, ax = plt.subplots()
for i in range(num_partVC):
    dataTrain,dataTest = dataHousing.validacionCruzada(i,particiones)
    model = RedNeuronal(LAYERS,ActivationFunctions,)
    optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate,weight_decay=0.01)
    LOSS=[]
    trainLoader =DataLoader(dataset=dataTrain,batch_size=5)
    LOSS,epoch,min_loss = model.train(dataHousing,optimizer,criterion,trainLoader,200)
    model_best = RedNeuronal(LAYERS,ActivationFunctions,"xavier")
    model_best.load_state_dict(torch.load('best_model.pt'))
    ACC.append(model_best.predict(dataTest, criterion))
    output = model_best(dataTest[0:-1][0]).detach().numpy()
    labels = np.array(dataTest[0:-1][1])
    output = np.clip(output,5,50)
    ax.scatter(labels, output, color="blue")
    ax.plot([labels.min(), labels.max()], [labels.min(), labels.max()], 'k--', lw=3)
    ax.set_xlabel('Real Values')
    ax.set_ylabel('Predicted Values')
    r2.append(r2_score(labels,output))
    print("iteracion", i + 1, ": perdida:", ACC[-1],"r2: ", r2[-1])

print("perdida media: ", sum(ACC)/num_partVC, " y r2 medio: ", sum(r2)/num_partVC)


plt.show()

#red neuronal profunda
print("Red Neuronal una capa oculta")

ACC=[]
r2=[]
learning_rate=0.01
```

```
LAYERS=[13,20,1]
ActivationFunctions = [F.relu]
criterion=torch.nn.MSELoss()
fig, ax = plt.subplots()

for i in range(num_partVC):
    dataTrain,dataTest = dataHousing.validacionCruzada(i,particiones)
    model = RedNeuronal(LAYERS,ActivationFunctions,"uniform")
    optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate,weight_decay=0.01)
    LOSS=[]
    trainLoader =DataLoader(dataset=dataTrain,batch_size=50)
    LOSS,epoch,min_loss = model.train(dataHousing,optimizer,criterion,trainLoader,200)
    model_best = RedNeuronal(LAYERS,ActivationFunctions)
    model_best.load_state_dict(torch.load('best_model.pt'))
    ACC.append(model_best.predict(dataTest, criterion))
    output = model_best(dataTest[0:-1][0]).detach().numpy()
    labels = np.array(dataTest[0:-1][1])
    output = np.clip(output,5,50)
    ax.scatter(labels, output, color="green")
    ax.plot([labels.min(), labels.max()], [labels.min(), labels.max()], 'k--', lw=3)
    ax.set_xlabel('Real Values')
    ax.set_ylabel('Predicted Values')
    r2.append(r2_score(labels,output))
    print("perdida en iteracion", i + 1, ":", ACC[-1],"r2: ", r2[-1])

print("perdida media: ", sum(ACC)/num_partVC, " y r2 medio: ", sum(r2)/num_partVC)
plt.show()
```

A.3. Código problema clasificación simple

```
from Data import Data
from RedNeuronal import RedNeuronal
import torch
import torch.nn.functional as F
import numpy as np
from torch.utils.data import DataLoader,random_split
from EstrategiaParticionado import ValidacionSimple,ValidacionCruzada
import matplotlib.pyplot as plt


#cargo los datos de con los que voy a trabajar
dataDiabetes = Data("../../data/pima-indians-diabetes.csv",0,8)
dataDiabetes.normalizar()

print(dataDiabetes.x[0])

torch.manual_seed(1)
#normalizamos los datos de entrada
```

```
num_partVC = 12
lenVC = int(len(dataDiabetes)/num_partVC)
print(len(dataDiabetes),lenVC)

#defino el criterio que se va a usar(crossEntropy) y el optimizador(descenso por
    gradientte)
learning_rate=0.01
LAYERS=[8,12,8,1]
ActivationFunctions = [F.relu,F.relu,torch.sigmoid]
criterion=torch.nn.BCELoss()
ACC = []
lengths = ()
for i in range(num_partVC):
    lengths = lengths + (lenVC,)
#usamos validacion cruzada para validar los datos
particiones = random_split(dataDiabetes, lengths)
acc_max = 0
cm_porcentaje = np.array([[0,0],[0,0]])
cm_numeros = np.array([[0,0],[0,0]])
for i in range(num_partVC):
    dataTrain,dataTest = dataDiabetes.validacionCruzada(i,particiones)
    model = RedNeuronal(LAYERS,ActivationFunctions,"xavier")
    optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate,weight_decay=0.01)
    LOSS=[]
    trainLoader =DataLoader(dataset=dataTrain,batch_size=10)
    LOSS,epoch,min_loss = model.train(dataDiabetes,optimizer,criterion,trainLoader,200)
    model_best = RedNeuronal(LAYERS,ActivationFunctions,"Normal")
    model_best.load_state_dict(torch.load('best_model.pt'))
    acc,cm = model_best.scoreSigmoid(dataTest)
    cm_porcentaje = cm_porcentaje + cm/lenVC
    cm_numeros = cm_numeros + cm
    ACC.append(acc)
    print("Acierto interacion ", i, ":",  ACC[-1]*100, " %")
    if acc > acc_max:
        torch.save(model_best.state_dict(), 'modelo_max_acc.pt')

print("la matriz de confusion con los porcentajes medios de acierto de la validacion
    cruzada: ", cm_porcentaje/num_partVC)
print("la matriz de confusion de la validacion cruzada: ", cm_numeros)
plt.plot(LOSS)
plt.tight_layout()
plt.xlabel("Epoch/Iterations")
plt.ylabel("Cost")
plt.show()
```

A.4. Código problema clasificación compleja

A.4.1. Código problema en PyTorch

```
import torch
from RedConvolucional import CNN
import torchvision.transforms as transforms
import torchvision.datasets as dsets
import matplotlib.pylab as plt
import numpy as np
import time
from sklearn.metrics import confusion_matrix

IMAGE_SIZE = 28
composed = transforms.Compose([transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)), transforms.
    ToTensor()])

train_dataset = dsets.MNIST(root='./data', train=True, download=True, transform=
    composed)
test_dataset = dsets.MNIST(root='./data', train=False, download=True, transform=
    composed)

print(len(train_dataset),len(test_dataset))

model = CNN(out_1=16, out_2=32)
criterion = torch.nn.CrossEntropyLoss()
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=200)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=5000)
start=time.time()
model.train(train_dataset,optimizer, criterion, train_loader, 10)
print("Tiempo de entrenamientoo en PyTorch: ",(time.time()-start)/60)
correct = 0

for x_test, y_test in test_loader:
    z = model.predict(x_test)
    _, yhat = torch.max(z.data, 1)
    cm = confusion_matrix(y_test,yhat)
    correct += (yhat == y_test).sum().item()
accuracy = correct / len(test_dataset)

print(accuracy)
print(cm)
```

A.4.2. Código problema en Keras

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
```

```
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
import time


def baseline_model():
    # creacion del modelo
    model = Sequential()
    model.add(Convolution2D(16, (5, 5), border_mode='valid', input_shape=(1, 28, 28),
              activation='relu', data_format='channels_first'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(32, 5, 5, border_mode='valid', input_shape=(16, 28, 28),
              activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(32*7*7, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # compilacion del modelo
    model.compile(loss='categorical_crossentropy', optimizer='sgd',
                  metrics=['accuracy'])
    return model

# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)
# carga del dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
#normalizacion de los valores de entrada de 0-255 a 0-1
X_train = X_train / 255
X_test = X_test / 255
# codificacion de la variable de salida
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
model = baseline_model()
# ajuste del modelo
start=time.time()
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=50, batch_size
    =200,
verbose=2)
print("Tiempo de entrenamientoo en Keras: ",(time.time()-start)/60)
# evaluacion del modelo
```

```
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f % %" % (100-scores[1]*100))
```


UAM
UNIVERSIDAD AUTONOMA
DE MADRID